Module 3: Stack ADT

Dr. Natarajan Meghanathan
Professor of Computer Science
Jackson State University
Jackson, MS 39217
E-mail: natarajan.meghanathan@jsums.edu
Stack ADT

• Features (Logical View)
  – A List that operates in a Last In First Out (LIFO) fashion
  – Insertion and deletion can be performed only from one end (i.e., the top of the stack)
    • The last added item has to be removed first
  – Operations:
    • Push( ) – adding an item to the top of the stack
    • Pop( ) – delete the item from the top of
    • Peek( ) – read the item in the top of the stack
    • IsEmpty( ) – whether there is any element in the top of the stack
  – All the above operations should be preferably implemented in O(1) time.
Dynamic Array-based Implementation of Stack ADT

- **List ADT**
  - Member variables
    - int *array
    - int maxSize
    - int endOfArray
  - **Constructor**
    - List(int size)
  - **Member functions**
    - bool isEmpty()
    - void resize(int s)
    - void insert(int data)
    - void insertAtIndex(int insertIndex, int data)
    - int read(int index)
    - void modifyElement(int index, int data)
    - void deleteElement(int deleteIndex)

- **Stack ADT**
  - Member variables
    - int *array
    - int maxSize
    - int topOfStack
  - **Constructor**
    - Stack(int size)
  - **Member functions**
    - bool isEmpty()
    - void resize(int s)
    - void push(int data)
    - int peek()
    - int pop()
Code 3.1: Dynamic Array-based Implementation of Stack ADT

C++

```cpp
private:
  int *array;
  int maxSize;
  int topOfStack; // same as endOfArray

public:
  Stack(int size){
    array = new int[size];
    maxSize = size;
    topOfStack = -1;
  }

  bool isEmpty(){
    if (topOfStack == -1)
      return true;

    return false;
  }
```

Java

```java
private int array[];
private int maxSize;
private int topOfStack;

public Stack(int size){
  array = new int[size];
  maxSize = size;
  topOfStack = -1;
}

public boolean isEmpty(){
  if (topOfStack == -1)
    return true;

  return false;
}
```
Code 3.1 (C++): Dynamic Array-based Implementation of Stack ADT

```cpp
void resize(int s) {
    int *tempArray = array;

    array = new int[s];

    for (int index = 0; index < min(s, topOfStack+1); index++) {
        array[index] = tempArray[index];
    }
    maxSize = s;
}

void push(int data) {  // same as insert 'at the end'
    if (topOfStack == maxSize-1)
        resize(2*maxSize);

    array[++topOfStack] = data;
}
```
Code 3.1 (Java): Dynamic Array-based Implementation of Stack ADT

public void resize(int s){
    int tempArray[] = array;

    array = new int[s];

    for (int index = 0; index < Math.min(s, topOfStack+1); index++){
        array[index] = tempArray[index];
    }

    maxSize = s;
}

public void push(int data){ // same as insert 'at the end'
    if (topOfStack == maxSize-1)
        resize(2*maxSize);

    array[++topOfStack] = data;
}
Code 3.1 (C++): Dynamic Array-based Implementation of Stack ADT

```c++
int peek()
{
    if (topOfStack >= 0)
        return array[topOfStack];
    else
        return -1000000;
    // an invalid value to indicate empty stack
}

int pop()
{
    if (topOfStack >= 0){
        return array[topOfStack--];
        // the topOfStack is decreased by one after
        // the value is retrieved
    }
    else
        return -1000000;
    // an invalid value to indicate empty stack
}
```
Code 3.1 (Java): Dynamic Array-based Implementation of Stack ADT

```java
public int peek(){
    if (topOfStack >= 0)
        return array[topOfStack];
    else
        return -1000000; // an invalid value indicating
                        // stack is empty
}
```

```java
public int pop(){
    if (topOfStack >= 0){
        return array[topOfStack--];
        // the topOfStack is decreased by one
    }
    else
        return -1000000; // an invalid value indicating
                        // stack is empty
}
```
Implementation of Stack

Dynamic Array vs. Singly/Doubly Linked List

• Push
  – Array: $O(n)$ time, due to need for resizing when the stack gets full
  – Singly Linked List: $\Theta(n)$ time
  – Doubly Linked List: $O(1)$ time

• Pop
  – Array: $O(1)$ time
  – Singly Linked List: $\Theta(n)$ time
  – Doubly Linked List: $O(1)$ time

• Peek
  – Array: $O(1)$ time
  – Singly Linked List: $\Theta(n)$ time
  – Doubly Linked List: $O(1)$ time

• A singly linked list-based implementation would be the most time consuming, as we would need to traverse the entire list for every push, pop and peek operation.
Code 3.2: Doubly Linked List-based Implementation of Stack

private:  
\begin{verbatim}
int data;
Node* nextNodePtr;
Node* prevNodePtr;
\end{verbatim}

public:
\begin{verbatim}
Node( )
void setData(int)
int getData()
void setNextNodePtr(Node* )
Node* getNextNodePtr( )
void setPrevNodePtr(Node* )
Node* getPrevNodePtr( )
\end{verbatim}

\hspace{1cm}

Class Stack (C++)

private:
\begin{verbatim}
Node* headPtr;
Node* tailPtr;
\end{verbatim}

public:
\begin{verbatim}
Stack()
    headPtr = new Node();
    tailPtr = new Node();
    headPtr->setNextNodePtr(0);
    tailPtr->setPrevNodePtr(0);

Node* getHeadPtr(){
    return headPtr;
}

Node* getTailPtr(){
    return tailPtr;
}

bool isEmpty(){
    if (headPtr->getNextNodePtr() == 0)
        return true;
    return false;
}
\end{verbatim}
Code 3.2: Doubly Linked List-based Implementation of Stack

Class Node (Java) Overview

```java
private int data;
private Node nextNodePtr;
private Node prevNodePtr;

public Node() {
    // Constructor
}
public void setData(int) {
    // Method to set data
}
public int getData() {
    // Method to get data
}
public void setNextNodePtr(Node) {
    // Method to set next node pointer
}
public Node getNextNodePtr() {
    // Method to get next node pointer
}
public void setPrevNodePtr(Node) {
    // Method to set prev node pointer
}
public Node getPrevNodePtr() {
    // Method to get prev node pointer
}
```

Class Stack (Java)

```java
class Stack{

    private Node headPtr;
    private Node tailPtr;

    public Stack(){
        headPtr = new Node();
        tailPtr = new Node();
        headPtr.setNextNodePtr(null);
        tailPtr.setPrevNodePtr(null);
    }

    public Node getHeadPtr(){
        return headPtr;
    }

    public Node getTailPtr(){
        return tailPtr;
    }

    public boolean isEmpty(){
        if (headPtr.getNextNodePtr() == null)
            return true;
        return false;
    }

    // Other methods...
}
```
Push Operation

Scenario 1: There is no node currently in the stack

Before Push

Head Node
- headPtr

Tail Node
- tailPtr

After Push

Head Node
- headPtr

New Node
- newNodePtr

Tail Node
- tailPtr
Push Operation

Scenario 2: There is at least one node already in the stack

// Before the new node is pushed, the prevNodePtr for the “tail node”
// would be pointing to the last node in the stack and the nextNodePtr
// for that last node would be pointing to NULL.
void push(int data) {
    Node* newNodePtr = new Node();
    newNodePtr->setData(data);
    newNodePtr->setNextNodePtr(0); // 1

    Node* lastNodePtr = tailPtr->getPrevNodePtr();

    if (lastNodePtr == 0) { // There is no other node in the Stack (Scenario 1)
        headPtr->setNextNodePtr(newNodePtr); // 2
        newNodePtr->setPrevNodePtr(0); // 3
    } else { // There is at least one node already in the Stack (Scenario 2)
        lastNodePtr->setNextNodePtr(newNodePtr); // 4
        newNodePtr->setPrevNodePtr(lastNodePtr); // 5
    }

    tailPtr->setPrevNodePtr(newNodePtr); // 6
    Whatever be the case, the prevNodePtr for the tail node will point to the newly pushed node
public void push(int data) {

    Node newNodePtr = new Node();
    newNodePtr.setData(data);
    newNodePtr.setNextNode(newNodePtr); \[1\]

    Node lastNodePtr = tailPtr.getPrevNode();

    if (lastNodePtr == null) {
        headPtr.setNextNode(newNodePtr); \[2\]
        newNodePtr.setPrevNode(null); \[3\]
    } else {
        lastNodePtr.setNextNode(newNodePtr); \[4\]
        newNodePtr.setPrevNode(lastNodePtr); \[5\]
    }

    tailPtr.setPrevNode(newNodePtr); \[6\]
}

// There is no other node in the Stack (Scenario 1)
// There is at least one node already in the Stack (Scenario 2)

Whatever be the case, the prevNodePtr for the tail node will point to the newly pushed node
Pop Operation

Scenario 1: There will be no node in the Stack after the Pop (i.e., there is just one node in the Stack before the Pop)

// Before Pop: The Head Node’s nextNodePtr and the Tail Node’s prevNodePtr are both pointing to the only node in the stack.
// After Pop: Both the Head Node’s nextNodePtr and the Tail Node’s prevNodePtr are set to NULL
Pop Operation

Scenario 2: There will be at least one node in the stack after the Pop operation is executed.
```cpp
int pop(){
    Node* lastNodePtr = tailPtr->getPrevNodePtr();
    Node* prevNodePtr = 0;

    int poppedData = -100000; //empty stack

    if (lastNodePtr != 0){ // If there is at least one node in the Stack before Pop
        prevNodePtr = lastNodePtr->getPrevNodePtr();
        poppedData = lastNodePtr->getData();
    }
    else // If the Stack is empty before pop, return an invalid value
        return poppedData;

    if (prevNodePtr != 0){ // If there is going to be at least one node in the Stack after the pop
        prevNodePtr->setNextNodePtr(0); // (Scenario 2)
        tailPtr->setPrevNodePtr(prevNodePtr);
    }
    else{ // If there is going to be no node in the Stack after the pop
        headPtr->setNextNodePtr(0); // (Scenario 1)
        tailPtr->setPrevNodePtr(0);
    }

    return poppedData;
}
```
public int pop(){
    Node lastNodePtr = tailPtr.getPrevNodePtr();
    Node prevNodePtr = null;

    int poppedData = -100000; //empty stack

    if (lastNodePtr != null){  // If there is at least one node in the Stack before Pop
        prevNodePtr = lastNodePtr.getPrevNodePtr();
        poppedData = lastNodePtr.getData();
    }
    else  // If the Stack is empty before Pop, return an invalid value
        return poppedData;

    if (prevNodePtr != null){  // If there is going to be at least one node in the
        prevNodePtr.setNextNodePtr(null);  // Stack after the pop
        tailPtr.setPrevNodePtr(prevNodePtr);  // (Scenario 2)
    }
    else{  // If there is going to be no node in the Stack after the pop
        headPtr.setNextNodePtr(null);  // (Scenario 1)
        tailPtr.setPrevNodePtr(null);
    }

    return poppedData;
}
Code 3.2: Peek Operation

C++

```cpp
int peek()
{
    Node* lastNodePtr = tailPtr->getPrevNodePtr();

    if (lastNodePtr != 0)
        return lastNodePtr->getData();
    else
        return -100000; // empty stack
}
```

Java

```java
public int peek()
{
    Node lastNodePtr = tailPtr.getPrevNodePtr();

    if (lastNodePtr != null)
        return lastNodePtr.getData();
    else
        return -100000; // empty stack
}
```
```cpp
#include <string>
#include <cstring>
#include <iostream>
#include <algorithm> // reverse
using namespace std;

int main(){

    string originalString;
    cout << "Enter a string: ";
    getline(cin, originalString);

    string upperCaseString(""");
    Initialize a new string as an empty string

    for (int index = 0; index < originalString.size(); index++){
        char c = originalString[index];
        upperCaseString += toupper(c);
        Getting the character at a specific index
        To get the uppercase version of a character
    }

    cout << upperCaseString << endl;
    reverse(originalString.begin(), originalString.end());
    reverse the string from its end to its beginning

    cout << "reversed string: " << originalString << endl;

    return 0;
}
```

This code reads a string (of possibly more than one Word) from the user and prints a new string that has the uppercase characters of the original string as well as reverses the string.
This code reads a string (of possibly more than one Word) from the user and prints a new string that has the uppercase characters of the original string as well as reverses the string.

```java
import java.util.*;

class StringProcessing{
    public static void main(String[] args){
        Scanner input = new Scanner(System.in);

        String originalString;
        System.out.print("Enter a string: ");
        originalString = input.nextLine();

        String upperCaseString = "";
        for (int index = 0; index < originalString.length(); index++){
            char c = originalString.charAt(index);
            upperCaseString += Character.toUpperCase(c);
        }

        System.out.println(upperCaseString);
        String reverseString = "";
        for (int index = originalString.length()-1; index >= 0; index--)
            reverseString += originalString.charAt(index);

        System.out.println("reverse string: "+reverseString);
    }
}
```

Note: String objects are not mutable in Java. Hence, we have to create a new String object that is the reverse of the original string.

To read more than word (a line) as string

Initialize a new string as an empty string

Getting the character at a specific index

To get the uppercase version of a character

Reverse the string from its end to its beginning
Parentheses Balance

• By parenthesis, we refer to the following symbols
  ( ), { }, [ ]

• The problem is about checking whether corresponding to each opening parenthesis there is a corresponding closing parenthesis in correct order.

• Examples for balanced parentheses
  – { [ ] ( ( ) ) }
  – [ ( { } ) ] [ ]
  – ( { } [ ( ) ] )

• Examples for unbalanced parentheses
  – [ [ ] ]
  – { ( ) [ ]}
Parentheses Balance (Program Logic)

• Logic to determine whether the parentheses in an expression are balanced or not. (We could use a Linked List or Dynamic Array-based Stack).
  – Input the expression as a string and read it one character at a time.
  – If the character read is a opening parenthesis, then push it into the stack
  – If the character read is a closing parenthesis, then pop the stack and check if the popped symbol is a matching opening parenthesis.
    • If so, continue.
    • Otherwise, stop and say, parenthesis is not balanced.
  – If the character read does not match with any of the above six symbols, then stop the program and say there is an invalid symbol in the input expression.
```cpp
int main()
{
    Stack stack;

    string expression;
    cout << "Enter an expression: ";
    cin >> expression;

    int index = 0;

    while (index < expression.size()){
        char symbol = expression[index];

        if (symbol == '{' || symbol == '(' || symbol == '['){
            stack.push(symbol);
            index++;
            continue;
        }
    }
```

Note: We will use the implementation of stack using doubly linked list. We will replace all the ‘int’ in the doubly linked list – based stack code to ‘char’ as appropriate.
else if (symbol == '}' || symbol == ')' || symbol == ']'){
    char topSymbol = stack.pop();
    if ((topSymbol == '{' && symbol == '})' ||
        (topSymbol == '(' && symbol == ')') ||
        (topSymbol == '[' && symbol == ']')){
        index++;
        continue;
    }
    else{
        cout << "parenthesis not balanced!!" << endl;
        return 0;
    }
}
else{
    cout << "Invalid symbol " << symbol << " in the expression!!" << endl;
    return 0;
}
}
cout << expression << " is balanced!!" << endl;
return 0;
Code 3.4 (Java): Parentheses Balancing

Note: We will use the implementation of stack using doubly linked list. We will replace all the ‘int’ in the single linked list – based stack code to ‘char’ as appropriate.
```java
else if (symbol == '}' || symbol == ')' || symbol == ']') {
    char topSymbol = stack.pop();
    if ((topSymbol == '{' && symbol == '}') ||
        (topSymbol == '(' && symbol == ')') ||
        (topSymbol == '[' && symbol == ']')) {
        index++;
        continue;
    }
    else {
        System.out.println("parenthesis not balanced!!");
        return;
    }
}
else {
    System.out.println("Invalid symbol " + symbol + " in the expression!!");
    return;
}
}

System.out.println(expression + " is balanced!!");
```
Example (C++) for String Tokenization
(breaking a string into tokens based on delimiters)

#include <iostream>
#include <string>
#include <cstring>
using namespace std;

int main(){

    string sample;

    cout << "Enter an expression: ";
    getline(cin, sample);

    char* sampleArray = new char[sample.length()+1];
    strcpy(sampleArray, sample.c_str());
    char* cptr = strtok(sampleArray, " ", "");

    int numSymbols = 0;
    int sumIntegers = 0;

    // for C-style string processing as character array
    
    In this example program, we will count the number of Symbols and the sum of the integers that appear in an input string ‘sample’

    Get a line of words as a string, sample
    Create a character array of size one more than the length of the string and copy the elements from the string ‘sample’ to the Character array ‘sampleArray’
    Set up a tokenizer for the character Array with , and blank space as Delimiters. The tokenizer will return Tokens as character arrays (strings)
while (cptr != 0){
    string token(cptr);
    if ((token.compare("@") == 0) ||
        (token.compare("!" ) == 0) ||
        (token.compare("#" ) == 0) ||
        (token.compare("$" ) == 0) ||
        (token.compare("%" ) == 0 )){
        numSymbols++;
    } else{
        int value = stoi(token);
        sumIntegers += value;
    }
    cptr = strtok(NULL, ", ", ) ;
}

cout << "number of operators: " << numSymbols << endl;
cout << "sum of the integers: " << sumIntegers << endl;
return 0;
}
Example (Java) for String Tokenization
(breaking a string into tokens based on delimiters)

Code 3.5

```java
import java.util.*;

// to use the StringTokenizer and Scanner class

class stringTokenizing{

    public static void main(String[] args) {

        String sample;
        Scanner input = new Scanner(System.in);

        System.out.print("Enter an expression: ");
        sample = input.nextLine();

        int numSymbols = 0;
        int sumIntegers = 0;

        StringTokenizer stk = new StringTokenizer(sample, ",", ";");

        // Get a line of words as a string, sample

        // Set up a tokenizer for the character Array with , and blank space as Delimiters. The tokenizer will return Tokens as strings
    }
}
```
while (stk.hasMoreTokens()) {
    String token = stk.nextToken();
    if ((token.equals("@")) || (token.equals("!")) || (token.equals("#")) || (token.equals("$")) || (token.equals("%"))) {
        numSymbols++;
    } else {
        int value = Integer.parseInt(token);
        sumIntegers += value;
    }
}
System.out.println("number of operators: " + numSymbols);
System.out.println("sum of the integers: " + sumIntegers);
Order of Operation
(Operator Precedence)

1) Parenthesis: ( ), { }, [ ]
2) Exponent: In case of a tie, we evaluate from right to left.
   Example: \(3^2^4 = 3^{16} = 43046721\)
3) Multiplication and Division: Break the tie, by evaluating from left to right.
4) Addition and Subtraction: Break the tie, by evaluating from left to right.

Example:
1) \(5 + 8 / 4 = 5 + 2 = 7\)
2) \(12 / 6 * 3 = 2 * 3 = 6\)
3) \(4 * 5 / 2 - 7 + 3\)
   \(= 20 / 2 - 7 + 3\)
   \(= 10 - 7 + 3\)
   \(= 3 + 3 = 6\)
4) \(4 * \{5 / (2 - 7) + 3\}\)
   \(= 4 * \{5 / (-5) + 3\}\)
   \(= 4 * \{-1 + 3\} = 8\)
Infix, Prefix and Postfix

- **Infix:** LeftOperand \(<\textit{Operator}>\) RightOperand
  - Example: 2 + 3
- **Prefix:** \(<\textit{Operator}>\) LeftOperand RightOperand
  - Example: + 2 3
- **Postfix:** LeftOperand RightOperand \(<\textit{Operator}>\)
  - Example: 2 3 +

Infix expressions use the order of operation to break the ties.
Prefix and Postfix expressions do not require the order of operation.
- In both prefix and postfix expressions, each operand will be associated only with one operator and hence no need to use rules of operator precedence.
- **For example:** consider a + b * c: this expression (infix notation) needs to use operator precedence for evaluation
  - + a * b c is the prefix notation and abc*+ is the postfix notation
Evaluation of Postfix Expression

Consider an infix expression: $A \times B + C \times D - E$
If evaluated in infix, the expression needs to be evaluated as follows:

$$(A \times B) + (C \times D) - E$$
$${(A \times B) + (C \times D)} - E$$

Converting this to postfix

$$(AB\times) + (CD\times) - E$$
$$(AB\times)(CD\times) + - E$$
$$(AB\times)(CD\times) + E -$$

Removing the parenthesis, the final postfix expression is: $AB\times CD\times + E -$ 

Evaluation Logic:
Scan the expression from left to right.
If we see an operand in the expression, push it into the stack.
If we see an operator, we pop the last two items from the stack, apply the operator on the two popped items (the first popped item will be the right operand and the second popped item will be the left operand) and push the result of the operation to the stack.
The only item in the stack after reading the entire expression is the value of the expression.
Evaluation of Post-Fix Expression

• Consider the post-fix expression
  AB*CD*+E –
• Let A = 2, B = 3, C = 1, D = 5, E = 4

<table>
<thead>
<tr>
<th>B = 3</th>
<th>D = 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>A = 2</td>
<td>C = 1</td>
</tr>
<tr>
<td></td>
<td>C*D = 5</td>
</tr>
<tr>
<td></td>
<td>A*B = 6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>E = 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A*B +</td>
</tr>
<tr>
<td>C*D = 11</td>
</tr>
<tr>
<td>A*B +</td>
</tr>
<tr>
<td>C*D – E = 7</td>
</tr>
</tbody>
</table>

Note: During a scan of a post-fix expression, the left operand of an operator goes first into the stack followed by the right operand. Hence, during a pop, the right operand comes first out of the stack, followed by the left operand
C++ Code for Postfix Evaluation

We will use the integer-based doubly linked list implementation of stack.

```cpp
Stack stack;

string expression;

cout << "Enter the expression to evaluate: ";
getline(cin, expression);
char* expressionArray = new char[expression.length()+1];
strcpy(expressionArray, expression.c_str());

char* cptr = strtok(expressionArray, "", "");
while (cptr != 0){
    string token(cptr);
    bool isOperator = false;
    if ( (token.compare("*") == 0) || (token.compare("/") == 0) ||
        (token.compare("+") == 0) || (token.compare("-") == 0) )
        isOperator = true;
    if (!isOperator){
        int val = stoi(token);
        stack.push(val);
    }
}
```

Check if the token is one of the four operators *, /, +, -; if so, set the ‘isOperator’ boolean to true.

If the token is not an operator, we assume it must be an integer, and push it into the Stack.
if (isOperator) {
    int rightOperand = stack.pop();
    int leftOperand = stack.pop();
    if (token.compare("\+\") == 0) {
        int result = leftOperand + rightOperand;
        cout << "intermediate result: " << result << endl;
        stack.push(result);
    } else if (token.compare("/\") == 0) {
        int result = leftOperand / rightOperand;
        cout << "intermediate result: " << result << endl;
        stack.push(result);
    } else if (token.compare("\-\") == 0) {
        int result = leftOperand - rightOperand;
        cout << "intermediate result: " << result << endl;
        stack.push(result);
    } else if (token.compare("\*\") == 0) {
        int result = leftOperand * rightOperand;
        cout << "intermediate result: " << result << endl;
        stack.push(result);
    }
    // end if
    cptr = strtok(NULL, ", ");
} // end while

cout << "final result: " << stack.pop() << endl;
return 0;

If ‘isOperator’ is true, then pop the top two integers from the Stack, perform the operation and Push the resulting value to the stack.

The right operand is popped first followed by the Left operand.

Set up the next iteration of the while loop by retrieving the next token.

The final value of the expression will be the only value in the stack when we exit the while loop.

Code 3.6 (C++) continued
Java Code for Postfix Evaluation

We will use the integer-based doubly linked list implementation of stack.

Code 3.6

Stack stack = new Stack();
String expression;
Scanner input = new Scanner(System.in);
System.out.print("Enter the expression to evaluate: ");
expression = input.nextLine();

StringTokenizer stk = new StringTokenizer(expression, ", ");
while (stk.hasMoreTokens()){

    String token = stk.nextToken();
    boolean isOperator = false;

    if ( (token.equals("*")) || (token.equals("/")) || (token.equals("+")) ||
        (token.equals("-")))
        isOperator = true;

    if (!isOperator){
        int val = Integer.parseInt(token);
        stack.push(val);
    }
}
If ‘isOperator’ is true, then pop the top two integers from the Stack, perform the operation and Push the resulting value to the stack.

The right operand is popped first followed by the Left operand.

Code 3.6 (Java) continued

```java
if (isOperator) {
    int rightOperand = stack.pop();
    int leftOperand = stack.pop();

    if (token.equals("*")) {
        int result = leftOperand * rightOperand;
        System.out.println("intermediate result: " + result);
        stack.push(result);
    } else if (token.equals("/")) {
        int result = leftOperand / rightOperand;
        System.out.println("intermediate result: " + result);
        stack.push(result);
    } else if (token.equals("+")) {
        int result = leftOperand + rightOperand;
        System.out.println("intermediate result: " + result);
        stack.push(result);
    } else if (token.equals("-")) {
        int result = leftOperand - rightOperand;
        System.out.println("intermediate result: " + result);
        stack.push(result);
    }
} //end if
} // end while

System.out.println("final result: " + stack.pop());
```

The final value of the expression will be the only value in the stack when we exit the while loop.
Evaluation of Prefix Expression

Consider an infix expression: A * B + C * D – E

If evaluated in infix, the expression needs to be evaluated as follows:

- (A * B) + (C * D) – E
- { (A * B) + (C * D)} – E

Converting this to prefix

- (*AB) + (*CD) – E
- + (*AB) (*CD) – E
- – + (*AB) (*CD) E

Removing the parenthesis, the final prefix expression is: – + *AB*CDE

Evaluation Logic:
Scan the expression from right to left (or reverse the expression and scan from left to right).

If we see an operand in the expression, push it into the stack.

If we see an operator, we pop the last two items from the stack, apply the operator on the two popped items (the first popped item will be the left operand and the second popped item will be the right operand) and push the result of the operation to the stack.

The only item in the stack after reading the entire expression is the value of the expression.
Evaluation of Pre-Fix Expression

• Consider the pre-fix expression
• $- + *AB*CDE$

Read this expression from right to left

• Let $A = 2$, $B = 3$, $C = 1$, $D = 5$, $E = 4$

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C*D</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A*B</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A<em>B + C</em>D</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A<em>B + C</em>D - E</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: During a scan of a pre-fix expression, the right operand of an operator goes first into the stack followed by the left operand. Hence, during a pop, the left operand comes first out of the stack, followed by the right operand.